



下载APP

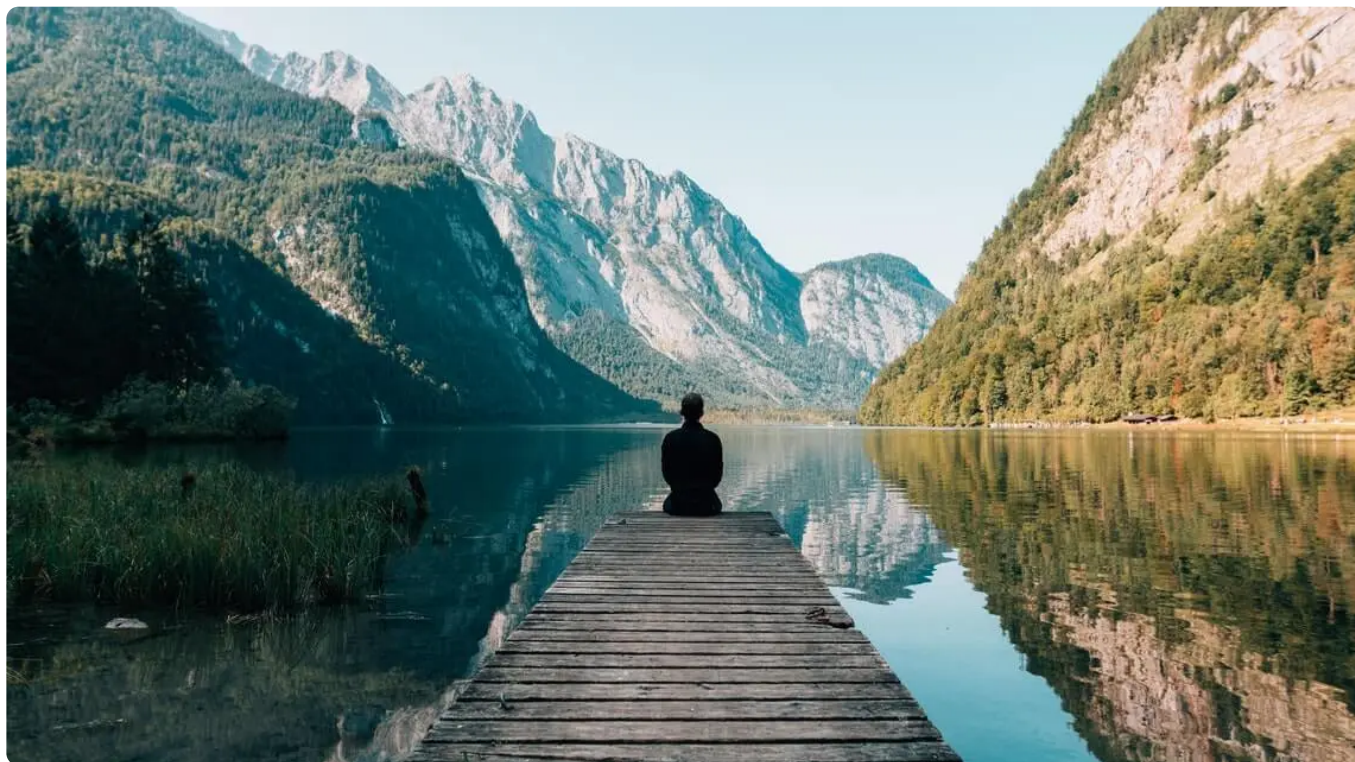


特别放送 | 给你一份Go项目中最常用的Makefile核心语法

2021-06-28 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 01:09 大小 1.06M



你好，我是孔令飞。今天，我们更新一期特别放送作为“加餐”，希望日常催更的朋友们食用愉快。

在第 [🔗 14 讲](#) 里，我强调了熟练掌握 Makefile 语法的重要性，还推荐你去学习陈皓老师编写的 [🔗 《跟我一起写 Makefile》 \(PDF 重制版\)](#)。也许你已经点开了链接，看到那么多 Makefile 语法，是不是有点被“劝退”的感觉？

其实在我看来，虽然 Makefile 有很多语法，但不是所有的语法都需要你熟练掌握，有些语法在 Go 项目中是很少用到的。要编写一个高质量的 Makefile，首先应该掌握一些核心的、最常用的语法知识。这一讲我就来具体介绍下 Go 项目中常用的 Makefile 语法规则，帮助你快速打好最重要的基础。



Makefile 文件由三个部分组成，分别是 Makefile 规则、Makefile 语法和 Makefile 命令（这些命令可以是 Linux 命令，也可以是可执行的脚本文件）。在这一讲里，我会介绍下 Makefile 规则和 Makefile 语法里的一些核心语法知识。在介绍这些语法知识之前，我们先看下如何使用 Makefile 脚本。

Makefile 的使用方法

在实际使用过程中，我们一般是先编写一个 Makefile 文件，指定整个项目的编译规则，然后通过 Linux make 命令来解析该 Makefile 文件，实现项目编译、管理的自动化。

默认情况下，make 命令会在当前目录下，按照 GNUmakefile、makefile、Makefile 文件的顺序查找 Makefile 文件，一旦找到，就开始读取这个文件并执行。

大多数的 make 都支持 “makefile” 和 “Makefile” 这两种文件名，但我建议使用 “**Makefile**”。因为这个文件名第一个字符大写，会很明显，容易辨别。make 也支持 -f 和 --file 参数来指定其他文件名，比如 make -f golang.mk 或者 make --file golang.mk。

Makefile 规则介绍

学习 Makefile，最核心的就是学习 Makefile 的规则。规则是 Makefile 中的重要概念，它一般由目标、依赖和命令组成，用来指定源文件编译的先后顺序。Makefile 之所以受欢迎，核心原因就是 Makefile 规则，因为 Makefile 规则可以自动判断是否需要重新编译某个目标，从而确保目标仅在需要时编译。

这一讲我们主要来看 Makefile 规则里的规则语法、伪目标和 order-only 依赖。

规则语法

Makefile 的规则语法，主要包括 target、prerequisites 和 command，示例如下：

```
1 target ...: prerequisites ...
2     command
3     ...
4     ...
```

[📄 复制代码](#)

target，可以是一个 object file（目标文件），也可以是一个执行文件，还可以是一个标签（label）。target 可使用通配符，当有多个目标时，目标之间用空格分隔。

prerequisites，代表生成该 target 所需要的依赖项。当有多个依赖项时，依赖项之间用空格分隔。

command，代表该 target 要执行的命令（可以是任意的 shell 命令）。

在执行 command 之前，默认会先打印出该命令，然后再输出命令的结果；如果不想打印出命令，可在各个 command 前加上@。

command 可以为多条，也可以分行写，但每行都要以 tab 键开始。另外，如果后一条命令依赖前一条命令，则这两条命令需要写在同一行，并用分号进行分隔。

如果要忽略命令的出错，需要在各个 command 之前加上减号-。

只要 targets 不存在，或 prerequisites 中有一个以上的文件比 targets 文件新，那么 command 所定义的命令就会被执行，从而产生我们需要的文件，或执行我们期望的操作。

我们直接通过一个例子来理解下 Makefile 的规则吧。

第一步，先编写一个 hello.c 文件。

[复制代码](#)

```
1 #include <stdio.h>
2 int main()
3 {
4     printf("Hello World!\n");
5     return 0;
6 }
```

第二步，在当前目录下，编写 Makefile 文件。

[复制代码](#)

```
1 hello: hello.o
2     gcc -o hello hello.o
3
```

```
4 hello.o: hello.c
5     gcc -c hello.c
6
7 clean:
8     rm hello.o
```

第三步，执行 make，产生可执行文件。

```
1 $ make
2 gcc -c hello.c
3 gcc -o hello hello.o
4 $ ls
5 hello  hello.c  hello.o  Makefile
```

 复制代码

上面的示例 Makefile 文件有两个 target，分别是 hello 和 hello.o，每个 target 都指定了构建 command。当执行 make 命令时，发现 hello、hello.o 文件不存在，就会执行 command 命令生成 target。

第四步，不更新任何文件，再次执行 make。

```
1 $ make
2 make: 'hello' is up to date.
```

 复制代码

当 target 存在，并且 prerequisites 都不比 target 新时，不会执行对应的 command。

第五步，更新 hello.c，并再次执行 make。

```
1 $ touch hello.c
2 $ make
3 gcc -c hello.c
4 gcc -o hello hello.o
```

 复制代码

当 target 存在，但 prerequisites 比 target 新时，会重新执行对应的 command。

第六步，清理编译中间文件。

Makefile 一般都会有一个 clean 伪目标，用来清理编译中间产物，或者对源码目录做一些定制化的清理：

```
1 $ make clean
2 rm hello.o
```

[复制代码](#)

我们可以在规则中使用通配符，make 支持三个通配符：*，? 和~，例如：

```
1 objects = *.o
2 print: *.c
3     rm *.c
```

[复制代码](#)

伪目标

接下来我们介绍下 Makefile 中的伪目标。Makefile 的管理能力基本上都是通过伪目标来实现的。

在上面的 Makefile 示例中，我们定义了一个 clean 目标，这其实是一个伪目标，也就是说我们不会为该目标生成任何文件。因为伪目标不是文件，make 无法生成它的依赖关系，也无法决定是否要执行它。

通常情况下，我们需要显式地标识这个目标为伪目标。在 Makefile 中可以使用 .PHONY 来标识一个目标为伪目标：

```
1 .PHONY: clean
2 clean:
3     rm hello.o
```

[复制代码](#)

伪目标可以有依赖文件，也可以作为“默认目标”，例如：

 复制代码

```
1 .PHONY: all
2 all: lint test build
```

因为伪目标总是会被执行，所以其依赖总是会被决议。通过这种方式，可以达到**同时执行所有依赖项**的目的。

order-only 依赖

在上面介绍的规则中，只要 prerequisites 中有任何文件发生改变，就会重新构造 target。但是有时候，我们希望**只有当 prerequisites 中的部分文件改变时，才重新构造 target**。这时，你可以通过 order-only prerequisites 实现。

order-only prerequisites 的形式如下：

 复制代码

```
1 targets : normal-prerequisites | order-only-prerequisites
2     command
3     ...
4     ...
```

在上面的规则中，只有第一次构造 targets 时，才会使用 order-only-prerequisites。后面即使 order-only-prerequisites 发生改变，也不会重新构造 targets。

只有 normal-prerequisites 中的文件发生改变时，才会重新构造 targets。这里，符号 “|” 后面的 prerequisites 就是 order-only-prerequisites。

到这里，我们就介绍了 Makefile 的规则。接下来，我们再来看下 Makefile 中的一些核心语法知识。

Makefile 语法概览

因为 Makefile 的语法比较多，这一讲只介绍 Makefile 的核心语法，以及 IAM 项目的 Makefile 用到的语法，包括命令、变量、条件语句和函数。因为 Makefile 没有太多复杂的语法，你掌握了这些知识点之后，再在实践中多加运用，融会贯通，就可以写出非常复杂、功能强大的 Makefile 文件了。

命令

Makefile 支持 Linux 命令，调用方式跟在 Linux 系统下调用命令的方式基本一致。默认情况下，make 会把正在执行的命令输出到当前屏幕上。但我们可以通过在命令前加@符号的方式，禁止 make 输出当前正在执行的命令。

我们看一个例子。现在有这么一个 Makefile：

```
1 .PHONY: test
2 test:
3     echo "hello world"
```

[复制代码](#)

执行 make 命令：

```
1 $ make test
2 echo "hello world"
3 hello world
```

[复制代码](#)

可以看到，make 输出了执行的命令。很多时候，我们不需要这样的提示，因为我们更想看的是命令产生的日志，而不是执行的命令。这时就可以在命令行前加@，禁止 make 输出所执行的命令：

```
1 .PHONY: test
2 test:
3     @echo "hello world"
```

[复制代码](#)

再次执行 make 命令：

```
1 $ make test
2 hello world
```

[复制代码](#)

可以看到，make 只是执行了命令，而没有打印命令本身。这样 make 输出就清晰了很多。

这里，**我建议**在命令前都加@符号，禁止打印命令本身，以保证你的 Makefile 输出易于阅读的、有用的信息。

默认情况下，每条命令执行完 make 就会检查其返回码。如果返回成功（返回码为 0），make 就执行下一条指令；如果返回失败（返回码非 0），make 就会终止当前命令。很多时候，命令出错（比如删除了一个不存在的文件）时，我们并不想终止，这时就可以在命令行前加 - 符号，来让 make 忽略命令的出错，以继续执行下一条命令，比如：

 复制代码

```
1 clean:
2     -rm hello.o
```

变量

变量，可能是 Makefile 中使用最频繁的语法了，Makefile 支持变量赋值、多行变量和环境变量。另外，Makefile 还内置了一些特殊变量和自动化变量。

我们先来看下最基本的**变量赋值**功能。

Makefile 也可以像其他语言一样支持变量。在使用变量时，会像 shell 变量一样原地展开，然后再执行替换后的内容。

Makefile 可以通过变量声明来声明一个变量，变量在声明时需要赋予一个初值，比如 `ROOT_PACKAGE=github.com/marmotedu/iam`。

引用变量时可以通过 `$()` 或者 `${}` 方式引用。我的建议是，用 `$()` 方式引用变量，例如 `$(ROOT_PACKAGE)`，也建议整个 makefile 的变量引用方式保持一致。

变量会像 bash 变量一样，在使用它的地方展开。比如：

 复制代码

```
1 GO=go
```

```
2 build:
3     $(GO) build -v .
```

展开后为：

```
1 GO=go
2 build:
3     go build -v .
```

[复制代码](#)

接下来，我给你介绍下 Makefile 中的 4 种变量赋值方法。

1. = 最基本的赋值方法。

例如：

```
1 BASE_IMAGE = alpine:3.10
```

[复制代码](#)

使用 = 进行赋值时，要注意下面这样的情况：

```
1 A = a
2 B = $(A) b
3 A = c
```

[复制代码](#)

B 最后的值为 c b，而不是 a b。也就是说，在用变量给变量赋值时，右边变量的取值，取的是最终的变量值。

2. := 直接赋值，赋予当前位置的值。

例如：

[复制代码](#)

```
1 A = a
2 B := $(A) b
3 A = c
```

B 最后的值为 a b。通过 `:=` 的赋值方式，可以避免 `=` 赋值带来的潜在的不一致。

3. `?=` 表示如果该变量没有被赋值，则赋予等号后的值。

例如：

```
1 PLATFORMS ?= linux_amd64 linux_arm64
```

[复制代码](#)

4. `+=`表示将等号后面的值添加到前面的变量上。

例如：

```
1 MAKEFLAGS += --no-print-directory
```

[复制代码](#)

Makefile 还支持**多行变量**。可以通过 `define` 关键字设置多行变量，变量中允许换行。定义方式为：

```
1 define 变量名
2 变量内容
3 ...
4 endef
```

[复制代码](#)

变量的内容可以包含函数、命令、文字或是其他变量。例如，我们可以定义一个 `USAGE_OPTIONS` 变量：

```
1 define USAGE_OPTIONS
2
```

[复制代码](#)

```
3 Options:
4     DEBUG      Whether to generate debug symbols. Default is 0.
5     BINS       The binaries to build. Default is all of cmd.
6     ...
7     V          Set to 1 enable verbose build. Default is 0.
8 endif
```

Makefile 还支持**环境变量**。在 Makefile 中，有两种环境变量，分别是 Makefile 预定义的环境变量和自定义的环境变量。

其中，自定义的环境变量可以覆盖 Makefile 预定义的环境变量。默认情况下，Makefile 中定义的环境变量只在当前 Makefile 有效，如果想向下层传递（Makefile 中调用另一个 Makefile），需要使用 `export` 关键字来声明。

下面的例子声明了一个环境变量，并可以在下层 Makefile 中使用：

```
1 ...
2 export USAGE_OPTIONS
3 ...
```

[复制代码](#)

此外，Makefile 还支持两种内置的变量：特殊变量和自动化变量。

特殊变量是 make 提前定义好的，可以在 makefile 中直接引用。特殊变量列表如下：

变量	含义
MAKE	当前make解释器的文件名
MAKECMDGOALS	命令行中指定的目标名（make的命令行参数）
CURDIR	当前make解释器的工作目录
MAKE_VERSION	当前make解释器的版本
MAKEFILE_LIST	make所需要处理的makefile文件列表，当前makefile的文件名总是位于列表的最后，文件名之间以空格进行分隔
.DEFAULT_GOAL	指定如果在命令行中未指定目标，应该构建哪个目标，即使这个目标不是在第一行
.VARIABLES	所有已经定义的变量名列表（预定义变量和自定义变量）
.FEATURES	列出本版本支持的功能，以空格隔开
.INCLUDE_DIRS	make查询makefile的路径，以空格隔开

Makefile 还支持**自动化变量**。自动化变量可以提高我们编写 Makefile 的效率和质量。

在 Makefile 的模式规则中，目标和依赖文件都是一系列的文件，那么我们如何书写一个命令，来完成从不同的依赖文件生成相对应的目标呢？

这时就可以用到自动化变量。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，一直到所有符合模式的文件都取完为止。这种自动化变量只应出现在规则的命令中。Makefile 中支持的自动化变量见下表。

变量	含义
\$@	表示规则中的目标文件集。在模式规则中，如果有多个目标，那么\$@就是匹配于目标中模式定义的集合
%	仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是foo.a(bar.o)，那么%就是bar.o，\$@就是foo.a。如果目标不是函数库文件（Unix下是.a，Windows下是.lib），那么，其值为空
\$<	依赖目标中的第一个目标名字。如果依赖目标是以模式（即%）定义的，那么\$<将是符合模式的一系列的文件集
\$?	所有比目标新的依赖目标的集合，以空格分隔
\$^	所有的依赖目标的集合，以空格分隔。如果在依赖目标中有多个重复的，那么这个变量会去除重复的依赖目标，只保留一份
\$+	这个变量很像\$^，也是所有依赖目标的集合，只是它不去除重复的依赖目标
\$	所有的order-only依赖目标的集合，以空格分割
\$*	这个变量表示目标模式中%及其之前的部分。如果目标是dir/a.foo.b，并且目标的模式是a.%b，那么，\$*的值就是dir/a.foo。

上面这些自动化变量中，\$*是用得最多的。\$* 对于构造有关联的文件名是比较有效的。如果目标中没有模式的定义，那么 \$* 也就不能被推导出。但是，如果目标文件的后缀是make 所识别的，那么 \$* 就是除了后缀的那一部分。例如：如果目标是 foo.c，因为.c 是make 所能识别的后缀名，所以 \$* 的值就是 foo。

条件语句

Makefile 也支持条件语句。这里先看一个示例。

下面的例子判断变量ROOT_PACKAGE是否为空，如果为空，则输出错误信息，不为空则打印变量值：

 复制代码

```
1 ifeq ($(ROOT_PACKAGE),)
2 $(error the variable ROOT_PACKAGE must be set prior to including golang.mk)
3 else
4 $(info the value of ROOT_PACKAGE is $(ROOT_PACKAGE))
5 endif
```

条件语句的语法为：

 复制代码

```
1 # if ...
2 <conditional-directive>
3 <text-if-true>
4 endif
5 # if ... else ...
6 <conditional-directive>
7 <text-if-true>
8 else
9 <text-if-false>
10 endif
```

例如，判断两个值是否相等：

 复制代码

```
1 ifeq 条件表达式
2 ...
3 else
4 ...
5 endif
```

ifeq 表示条件语句的开始，并指定一个条件表达式。表达式包含两个参数，参数之间用逗号分隔，并且表达式用圆括号括起来。

else 表示条件表达式为假的情况。

endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。

表示条件关键字，有 4 个关键字：ifeq、ifneq、ifdef、ifndef。

为了加深你的理解，我们分别来看下这 4 个关键字的例子。

1. ifeq：条件判断，判断是否相等。

例如：

```
1 ifeq (<arg1>, <arg2>)  
2 ifeq '<arg1>' '<arg2>'  
3 ifeq "<arg1>" "<arg2>"  
4 ifeq "<arg1>" '<arg2>'  
5 ifeq '<arg1>' "<arg2>"
```

 复制代码

比较 arg1 和 arg2 的值是否相同，如果相同则为真。也可以用 make 函数 / 变量替代 arg1 或 arg2，例如 ifeq (\$(origin ROOT_DIR),undefined) 或 ifeq (\$(ROOT_PACKAGE),)。origin 函数会在之后专门讲函数的一讲中介绍到。

2. ifneq：条件判断，判断是否不相等。

```
1 ifneq (<arg1>, <arg2>)  
2 ifneq '<arg1>' '<arg2>'  
3 ifneq "<arg1>" "<arg2>"  
4 ifneq "<arg1>" '<arg2>'  
5 ifneq '<arg1>' "<arg2>"
```

 复制代码

比较 arg1 和 arg2 的值是否不同，如果不同则为真。

3. ifdef：条件判断，判断变量是否已定义。

```
1 ifdef <variable-name>
```

 复制代码

如果值非空，则表达式为真，否则为假。也可以是函数的返回值。

4. ifndef : 条件判断，判断变量是否未定义。

```
1 ifndef <variable-name>
```

[复制代码](#)

如果值为空，则表达式为真，否则为假。也可以是函数的返回值。

函数

Makefile 同样也支持函数，函数语法包括定义语法和调用语法。

我们先来看下自定义函数。 make 解释器提供了一系列的函数供 Makefile 调用，这些函数是 Makefile 的预定义函数。我们可以通过 define 关键字来自定义一个函数。自定义函数的语法为：

```
1 define 函数名
2 函数体
3 endef
```

[复制代码](#)

例如，下面这个自定义函数：

```
1 define Foo
2     @echo "my name is ${0}"
3     @echo "param is ${1}"
4 endef
```

[复制代码](#)

define 本质上是定义一个多行变量，可以在 call 的作用下当作函数来使用，在其他位置使用只能作为多行变量来使用，例如：

```
1 var := $(call Foo)
```

[复制代码](#)

```
2 now := $(Foo)
```

自定义函数是一种过程调用，没有任何的返回值。可以使用自定义函数来定义命令的集合，并应用在规则中。

再来看下预定义函数。 刚才提到，make 编译器也定义了很多函数，这些函数叫作预定义函数，调用语法和变量类似，语法为：

```
1 $(<function> <arguments>)
```

[复制代码](#)

或者

```
1 ${<function> <arguments>}
```

[复制代码](#)

<function>是函数名，<arguments>是函数参数，参数间用逗号分割。函数的参数也可以是变量。

我们来看一个例子：

```
1 PLATFORM = linux_amd64
2 GOOS := $(word 1, $(subst _, ,$(PLATFORM)))
```

[复制代码](#)

上面的例子用到了两个函数：word 和 subst。word 函数有两个参数，1 和 subst 函数的输出。subst 函数将 PLATFORM 变量值中的 _ 替换成空格（替换后的 PLATFORM 值为 linux amd64）。word 函数取 linux amd64 字符串中的第一个单词。所以最后 GOOS 的值为 linux。

Makefile 预定义函数能够帮助我们实现很多强大的功能，在编写 Makefile 的过程中，如果有功能需求，可以优先使用这些函数。如果你想使用这些函数，那就需要知道有哪些函数，以及它们实现的功能。

常用的函数包括下面这些，你需要先有个印象，以后用到时再来查看。

函数名	功能描述
<code>\$(origin <variable>)</code>	告诉变量的“出生情况”，有如下返回值： - undefined: <variable> 从来没有定义过 - default: <variable> 是一个默认的定义 - environment: <variable> 是一个环境变量 - file: <variable> 这个变量被定义在 Makefile中 - command line: <variable> 这个变量是被命令行定义的 - override: <variable> 是被 override 指示符重新定义的 - automatic: <variable> 是一个命令运行中的自动化变量
<code>\$(addsuffix <suffix>,<names...>)</code>	把后缀<suffix>加到<names>中的每个单词后面，并返回加过后缀的文件名序列
<code>\$(addprefix <prefix>,<names...>)</code>	把前缀<prefix>加到<names>中的每个单词后面，并返回加过前缀的文件名序列
<code>\$(wildcard <pattern>)</code>	扩展通配符，例如： <code>\$(wildcard \${ROOT_DIR}/build/docker/*)</code>
<code>\$(word <n>,<text>)</code>	取字符串<text>中第<n>个单词（从1开始），并返回字符串<text>中第<n>个单词。如 <n>比<text>中的单词数要大，那么返回空字符串
<code>\$(subst <from>,<to>,<text>)</code>	把字符串 <text> 中的 <from> 字符串替换成 <to>，并返回被替换后的字符串
<code>\$(eval <text>)</code>	将<text>的内容将作为makefile的一部分而被make解析和执行
<code>\$(firstword <text>)</code>	取字符串 <text> 中的第一个单词，并返回字符串 <text> 的第一个单词
<code>\$(lastword <text>)</code>	取字符串 <text> 中的最后一个单词，并返回字符串 <text> 的最后一个单词
<code>\$(abspath <text>)</code>	将<text>中的各路径转换成绝对路径，并将转换后的结果返回
<code>\$(shell cat foo)</code>	执行操作系统命令，并返回操作结果
<code>\$(info <text ...>)</code>	输出一段信息
<code>\$(warning <text ...>)</code>	输出一段警告信息，而 make 继续执行
<code>\$(error <text ...>)</code>	产生一个致命的错误，<text ...> 是错误信息
<code>\$(filter <pattern>,<text>)</code>	以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式，返回符合模式<pattern>的单词。

<code><pattern...>,<text>)</code>	的单词。可以有多个模式。返回符合模式<pattern>的子串
<code>\$(filter-out <pattern...>,<text>)</code>	以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式，并返回不符合模式<pattern>的字串
<code>\$(dir <names...>)</code>	从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠 (/) 之后的部分。返回文件名序列<names>的非目录部分
<code>\$(notdir <names...>)</code>	从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠 (/) 之后的部分。返回文件名序列<names>的非目录部分
<code>\$(strip <string>)</code>	去掉<string>字串中开头和结尾的空字符，并返回去掉空格后的字符串
<code>\$(suffix <names...>)</code>	从文件名序列<names>中取出各个文件名的后缀。返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串
<code>\$(foreach <var>,<list>,<text>)</code>	把参数<list>中的单词逐一取出，放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次 <text>会返回一个字符串，循环过程中<text>所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值

引入其他 Makefile

除了 Makefile 规则、Makefile 语法之外，Makefile 还有很多特性，比如可以引入其他 Makefile、自动生成依赖关系、文件搜索等等。这里我再介绍一个 IAM 项目的 Makefile 用到的重点特性：引入其他 Makefile。

在 [🔗 14 讲](#) 中，我们介绍过 Makefile 要结构化、层次化，这一点可以通过**在项目根目录下的 Makefile 中引入其他 Makefile** 来实现。

在 Makefile 中，我们可以通过关键字 `include`，把别的 makefile 包含进来，类似于 C 语言的 `#include`，被包含的文件会插入在当前的位置。`include` 用法为 `include <filename>`，示例如下：

 复制代码

```
1 include scripts/make-rules/common.mk
2 include scripts/make-rules/golang.mk
```

include 也可以包含通配符 `include scripts/make-rules/*`。make 命令会按下面的顺序查找 makefile 文件：

1. 如果是绝对或相对路径，就直接根据路径 include 进来。
2. 如果 make 执行时，有 `-I` 或 `--include-dir` 参数，那么 make 就会在这个参数所指定的目录下去找。
3. 如果目录 `<prefix>/include` (一般是 `/usr/local/bin` 或 `/usr/include`) 存在的话，make 也会去找。

如果有文件没有找到，make 会生成一条警告信息，但不会马上出现致命错误，而是继续载入其他的文件。一旦完成 makefile 的读取，make 会再重试这些没有找到或是不能读取的文件。如果还是不行，make 才会出现一条致命错误信息。如果你想让 make 忽略那些无法读取的文件继续执行，可以在 include 前加一个减号 `-`，如 `-include <filename>`。

总结

在这一讲里，为了帮助你编写一个高质量的 Makefile，我重点介绍了 Makefile 规则和 Makefile 语法里的一些核心语法知识。

在讲 Makefile 规则时，我们主要学习了规则语法、伪目标和 order-only 依赖。掌握了这些 Makefile 规则，你就掌握了 Makefile 中最核心的内容。

在介绍 Makefile 的语法时，我只介绍了 Makefile 的核心语法，以及 IAM 项目的 Makefile 用到的语法，包括命令、变量、条件语句和函数。你可能会觉得这些语法学习起来比较枯燥，但还是那句话，工欲善其事，必先利其器。希望你能熟练掌握 Makefile 的核心语法，为编写高质量的 Makefile 打好基础。

今天的内容就到这里啦，欢迎你在下面的留言区谈谈自己的看法，我们下一讲见。

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

👍 赞 1

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 特别放送 | 给你一份清晰、可直接套用的Go编码规范

更多课程推荐

容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师
云平台架构师



涨价倒计时 🕒

今日订阅 **¥69**，7月20日涨价至 **¥129**

精选留言 (7)

💬 写留言



helloworld 🏆

2021-06-28

我觉得变量用\${}, 函数用\$(),这样能很好区分，对熟悉shell的人也更友好

作者回复: 听起来没毛病，保持一致即可



👍 4



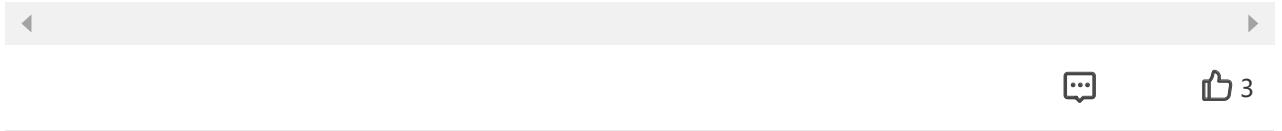
pedro

2021-06-28

周一就更新，值👍

展开 ▾

编辑回复: 这篇是额外的加餐，不占排期哦

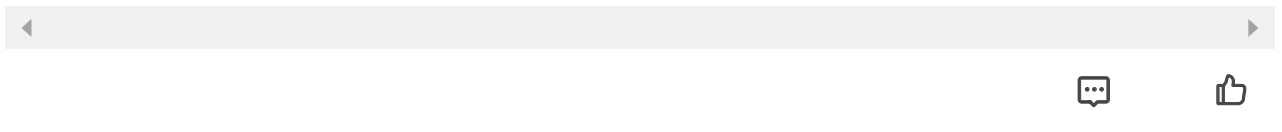


成楼

2021-07-16

由于缺少云上开发资源，本地虚拟机的环境配置能否给些建议，可以创建一个群方便交流吗？

作者回复: 可以看下IAM项目根目录下的README文档，有微信号



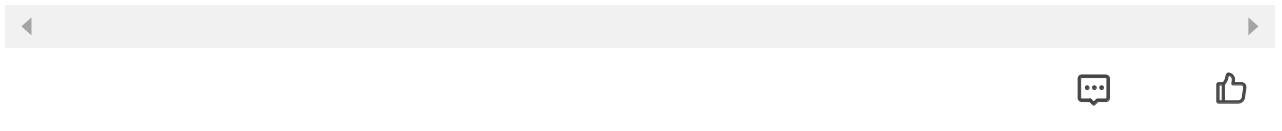
成楼

2021-07-16

有学员群吗？可以加入下吗？方便交流和提升，谢谢。

展开 ∨

作者回复: 可以看下iam仓库根目录下的README文档，最后有微信号



老老远

2021-07-12

感觉不错，希望以后再出些学完后实际可用的项目教学

展开 ∨



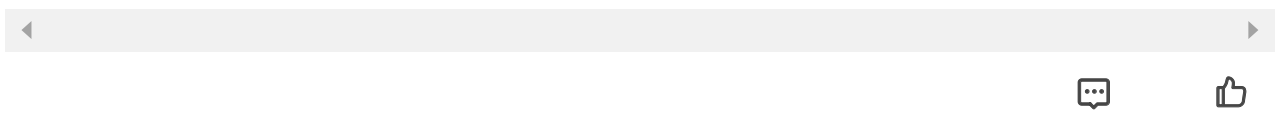
Realm

2021-07-10

老师，函数dir，nodir注释好像有笔误.

展开 ∨

作者回复: 没找到你说的哪个位置，可以再明确下位置吗，thanks



**Tiandh**

2021-06-28

老师，这两句话不理解

因为伪目标不是文件，make 无法生成它的依赖关系，也无法决定是否要执行它。

因为伪目标总是会被执行，所以其依赖总是会被决议。

作者回复: 就是伪目标总是会被执行

